 By Gary Peyrot, Lattice Semiconductor

USING AN EFFECTIVE TESTBENCH IS AN IMPORTANT PART OF PROGRAMMABLE-LOGIC SIMULATION. KNOWING WHICH VHDL FEATURES SUPPORT HIGH-LEVEL MODELING HELPS REDUCE YOUR TESTBENCH-DEVELOPMENT TIME.

Behavioral modeling in VHDL simulation

THE EFFECTIVE USE OF SIMULATION offers several advantages for product-development teams. The major benefits of effective simulation are quicker time to market, increased quality, and decreased risk. By offering enhanced controllability and observability, a well-designed simulation can test situations that are difficult to generate in a prototype debugging environment. You can easily check boundary conditions when you have complete control of the environment. Worst-case conditions may include the simultaneous occurrence of independent functions that are difficult to achieve in an actual system. You can set up these conditions in a controllable simulation.

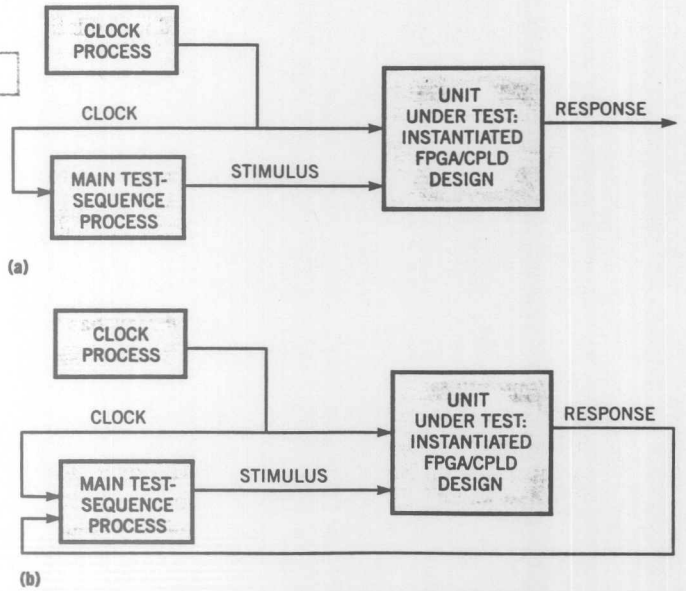
Simulation also enables more extensive observations of your design. Internal signals are available, regardless of whether spare I/O pins are on the device. You can even more easily probe external signals during simulation. With a few keystrokes or mouse-clicks, you can use a good logic simulator to probe a 64-bit data bus in seconds; try that with a logic analyzer. It could take the better part of a day in the lab to get a complex design set up for monitoring.

These same advantages of controllability and observability allow you to more quickly and easily catch errors with simulation than with hardware debugging. Furthermore, you can change the design more quickly in a simulation than on a hardware prototype. This feature shortens the debugging cycle and eases integration, especially if you simulate different modules together before integration. Simulation not only decreases the debugging cycle, but also requires fewer prototype hardware-design changes. This feature is good news to anyone who has had to build a second prototype board because the design needed so many changes. Designing a new pc board easily adds weeks to a schedule. With higher level techniques, you can use simulation to test the feasibility of basic design approaches.

COMMON SIMULATION MISTAKES

A key question is, "What keeps simulations from being efficient?" The basic problem is often using low-level methods, such as waveform-vector entry, for large and complex designs. This technique, in

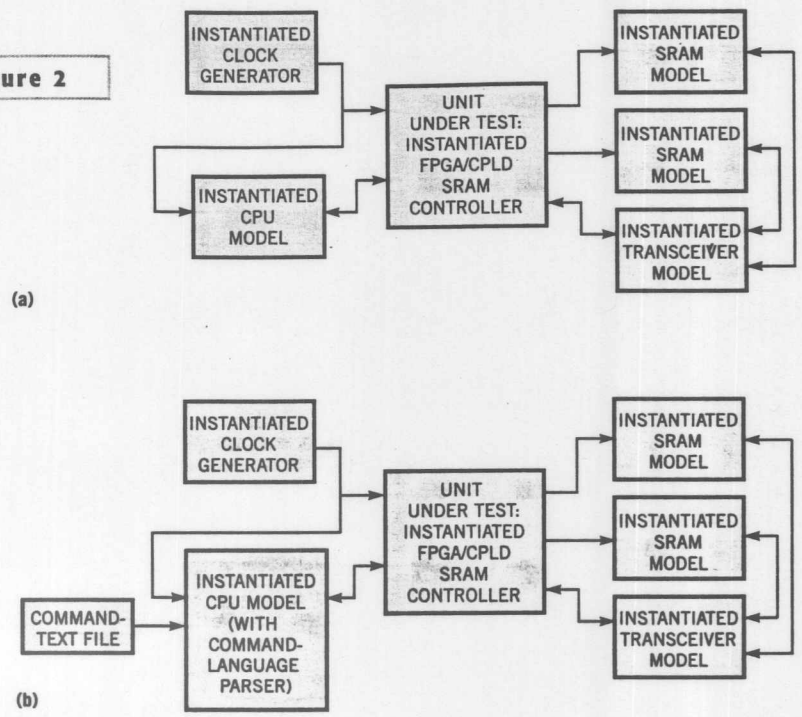
Figure 1



You can improve a simple testbench (a) by adding statements that check the responses of the unit under test and modifying the test sequence according to those responses (b).

turn, causes many of the classic failures in system simulation, including insufficient fault coverage and inadequate circuit-response characterization. In a large design, it is difficult to get proper fault coverage with low-level methods, which force you to manually generate each vector and test case. When vector generation is tedious and time-consuming, it may be difficult to specify all the test scenarios of interest. Even if you specify the test cases, it may be difficult to make them complete. Because most projects using simple techniques use visual inspection to qualify circuit responses, you may miss many visible-circuit malfunctions. More powerful simulation techniques actually simplify the task of design verification. Some people believe that the only methods of handling complexity are hierarchy and abstraction. Modeling makes use of both hierarchy and abstraction to allow the simulation of a complex system to be both manageable and efficient.

Figure 2



By using functional models in your testbench for the unit under test and the surrounding board-level components, you create a more efficient testbench, either with a fixed CPU model (a) or with a CPU model that you can modify with a command file (b).

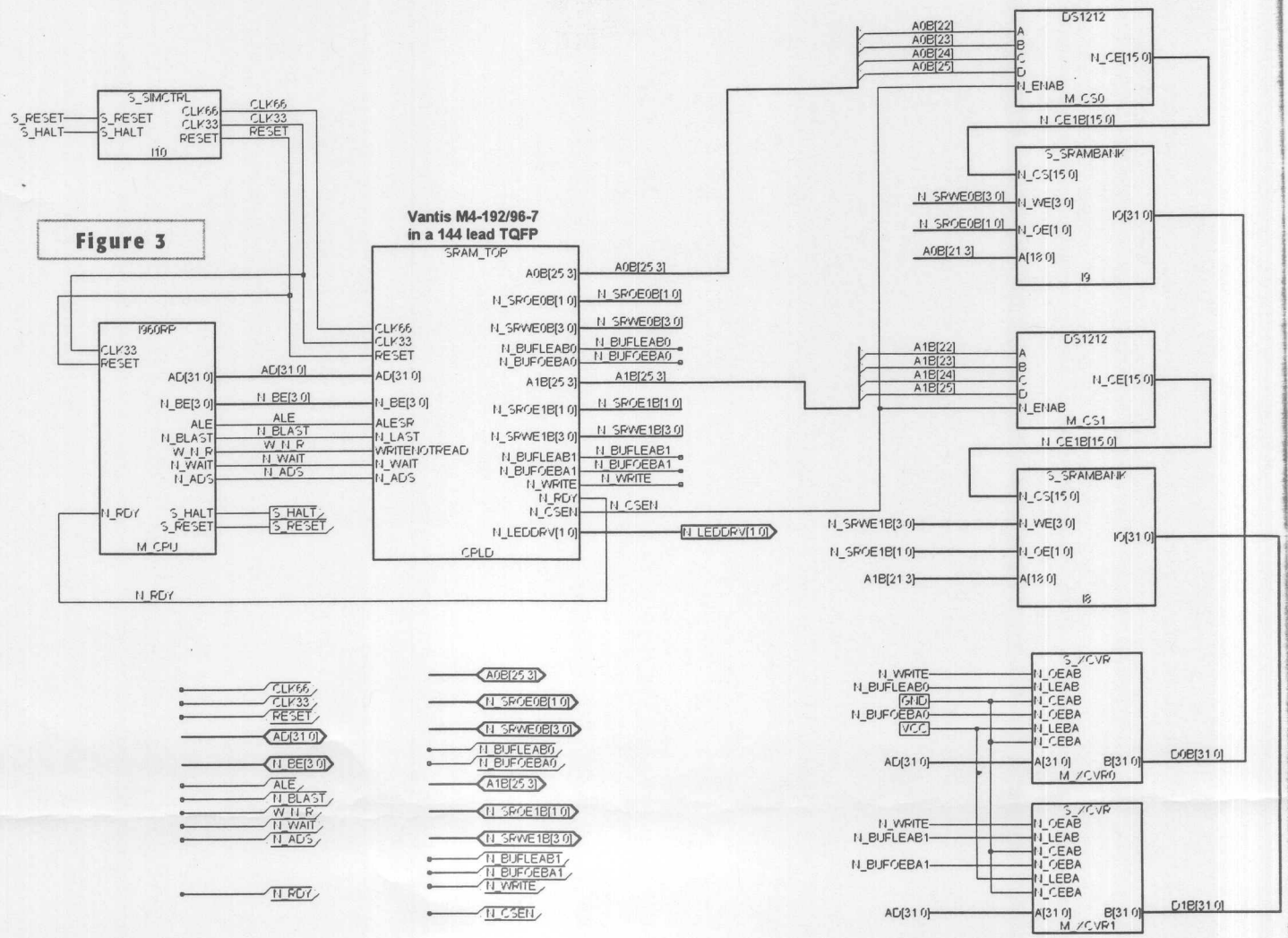
GOALS FOR A GOOD TESTBENCH

The key requirements of a good testbench are completeness, ease of use, flexibility, reusability, and runtime efficiency. These features are interrelated: A flexible design is easier to use, easier to complete, and to easier to reuse.

High fault coverage is essential for an effective simulation project. A good simulation strategy facilitates the development of a testbench that covers all pertinent operational scenarios. You can best accomplish this goal by using techniques that allow simulation designers to work at a fairly high level of abstraction. It is easier to follow the flow of a simulation if you can specify the test scenarios at a macro level rather than at the atomic level. A good simulation technique should support a high level of abstraction. With many features that support behavioral modeling at a high level, VHDL easily satisfies this requirement. Simulation methods based on modeling of system-level components naturally work at a high level of abstraction. For example, having a model of a CPU allows you to specify the types of bus cycles. Building the testbench around the CPU model allows you to set up the test patterns in a manner similar to that of a programmer writing a diagnostic program. At a high level, the simulation flows naturally by setting up DMA transfers, programmed I/O, and various bus operations. This approach allows an engineer to focus on the functional requirements of the test, leaving bit manipulation to the hardware model.

Although a simulation method should be easy to use, this strategy involves a trap: The methods that are easy to use for small designs do not scale well to more complex designs. Using a library of preverified functional models, you can easily build a testbench that has comprehensive timing and data checking. A good library has these features in the models themselves. It is also easy to write test cases because the "virtual system" of the testbench mimics the operation of the actual hardware. Anyone who understands the operation of the hardware system can understand the steps needed to exercise the design being tested.

A good simulation methodology must be flexible enough to allow changes to the simulation as the test system gains knowledge or test requirements change. If a testbench is equivalent to a pile of "spaghetti code," modifying the testbench is difficult. Object-oriented software techniques are available to offer this kind of flexibility to computer programmers. Embedding functions into functional-model components offers similar benefits to engineers designing complex chip or system simulation. To be easily modified, hardware designs should have strong cohesion and loose coupling. The control path in a design should be simple. By mimicking the structure of the system design in a testbench, you can easily duplicate the behavior of the system. If the system design changes, you can straightforwardly modify the testbench model to ensure that it matches the system. Modeling also helps flexibility in other ways. Defining functions associated with component models allows you to make changes in one place that can have effects everywhere you use the component. Contrast this with the global search-and-replace or cut-and-paste operations so common with simple testbenches. Proper model-



This top-level schematic of a CPLD-based SRAM-controller design shows how you link the controller to models for all the other major components in the design.

ing should yield flexible simulation architectures.

Design reuse is becoming a major factor in closing the productivity gap in deep-submicron VLSI design methodologies. High-density FPGAs and CPLDs that promise fast time to market require similar techniques. Because chips are inherently reusable, so are the simulation models that mimic their behavior. Just as designers habitually use many of the same components in many designs, you can also reuse good simulation models.

A good testbench must execute quickly. Modeling has no inherent advantage over low-level methods. Just as a good assembly-language program is generally faster than one written in a high-level language, a low-level testbench is probably faster than a corresponding modeling simulation. For complex simulations, the advantages of modeling far outweigh a decrease in simulation speed. The execution time for a typical FPGA is smaller than the time spent developing the simulation suite. Besides, when you properly partition a simulation according to functional entities, or models, it may be easier to isolate the time-consuming portions of a simulation and optimize these portions. In general, a well-designed modeled testbench does not have to be significantly slower than a low-level testbench. Furthermore, although individual simulation cycles in low-level modeling may execute faster, the overall objective of providing a thoroughly simulated design is faster using higher level techniques, because fault coverage is more effective and less prone to error. In other words, using lower level techniques takes much longer to get the fault coverage that higher level techniques provide.

Figure 1a shows a block diagram of a typical simple testbench. In the code example of **Listing 1**, the code structure is simple and flat. The unit under test is instantiated as a component, and one process controls the main test sequence. Inline code generates circuit stimuli, and multiple wait statements control the main sequence timing. Alternatively, you could use multiple *after* conditions in a transport statement to control timing. Most novice users of VHDL simulation validate circuit responses by inspecting the resulting waveforms. Such a test-

TABLE 1—VHDL FEATURES THAT SUPPORT MODELING

Feature	Application	Benefits
Multiple architectures	Implementing structures to aid simulation	Some difficult-to-simulate structures, such as wide counters, can cause problems for a simulation; use multiple architectures to instantiate a different counter that will reach terminal count much sooner than the counter actually needs for the application.
	Gate-level versus RTL or behavioral models	You can use multiple architectures to allow a testbench to instantiate the presynthesis RTL model or the postfit gate-level model, depending upon the VHDL configuration associated with the project
	Alternative implementations for stepwise refinement	The first or simplest simulation model may not be the most efficient. You can switch alternative models into and out of the simulation to compare results. For example, you could replace the original CPU model with a newer model that interfaces to a text-file language parser.
Attributes	VHDL attributes extract information from signals and other objects	Attributes are extremely useful in simulations, especially for checking functional and timing operation. You attach them to a prefix, as in the case of the familiar "clk'event." Table 2 lists the most useful attributes.
Sequential control structures	if-then-else	These statements are useful for synthesis and simulation for describing complex, arbitrary conditions for controlling logic. Although these statements can be tricky for synthesis because complex statements can generate large amounts of logic, simulation does not suffer because testbenches do not synthesize into hardware.
	case	As in synthesis, case statements are useful for structured control, functioning as multiplexers when you synthesize them or use them in testbenches.
	for loop	For loops allow for a number of iterations, and you can use the index variable to sequence through indexed arrays or other data structures. They are useful for applying pattern vectors from a text file or internal array to a design under test.
	while loop	While loops specify the conditions under which the loop will continue to execute. They are useful for simulations, but synthesis does not widely support them.
	infinite loop	An infinite loop is one that has no specified terminating condition and usually includes an exit condition. An infinite loop can cause the simulator to "hang" if a run command is used without a limit.
	exit	The exit statement causes any loop to terminate if the exit condition specified is satisfied.
Delay statements	wait for (*time*) wait until (*condition*) wait on (*sensitivity list*) Transport statements	This statement suspends execution until the specified time has elapsed. This statement suspends execution until a specified condition is true. This statement suspends execution until an event occurs on one of the signals listed in the sensitivity list. Transport statements affect only individual signals.
Assert statements	Response checking and message generation	Assert statements generate messages based on whether a specified condition is true; they are useful with self-checking models using the signal attributes to check behavior.
Component instantiation	Instantiation of models into structural code	Models are simply instantiated in a design as components. You use signals to connect the models to the design under test, just as if they were actual components on a board.
Generics	Design reuse for generic models	Generics allow the passing of a numeric parameter, or instance-specific information, into an entity. This feature allows for parameterized models, which are more flexible in their application or reuse. For example, you could develop a variable-size and -width SRAM model. Generics also allow you to pass delays into each instance of a component's instantiation.
Arrays and user-defined types	Data structures for pattern generation or checking	Arrays are useful for many reasons. For example, they let you sequence through a list of vectors or patterns for application to a circuit.
Packages	Systemwide definitions	Packages allow you to define the key parameters of a test program in a common file. This feature is useful for any type of complex design, especially for common definitions for use by a team of engineers.

bench typically does little or no qualification of circuit performance.

The lack of functional partitioning in this testbench makes it difficult to implement complex controls. For example, you may want to vary control-signal timing to verify that the unit under test can tolerate all the allowed variations on its inputs. An examination of the testbench code shows that a single wait statement, *wait for clk_period*, affects the timing of all subsequent signal assignments. It can be difficult to manage the timing of the various stimulus signals when their timing is interactive. This situation becomes worse when someone copies and pastes these wait statements multiple times into a large process. The more complex the design, the less adequate these methods become. Static-timing schemes often cannot handle interactive chip interfaces. The need to implement such handshaking is often what first drives engineers to consider modeling. A process can use conditional sequential statements, *if-then-else* or *case* statements, that actively interact with the unit under test. Although you can perform this handshaking without building a model, you would not unify structure and function. The discipline of model building is helpful because it immediately suggests that a testbench's function should mimic the interactive nature of the system under test.

LISTING 1—USEFUL FILE-I/O COMMANDS

```
-- main stimulus process
   p_stim: process
   begin
-- main stimulus process
   p_stim: process begin
      csn <= '1';
      wen <= '1';
      end_in <= (others => '0');
      rst <= '1';
      wait for clk_period*2;
      rst <= '0';
      wait for clk_period;
      assert (out_bus = x"5") report "Out did not reach count 5"
         severity error;
         waitn <= '0';
         wait for clk_period*2;
      waitn <= '1';
      wait for clk_period*20;
      assert false
         report "Reached end of stimulus."
            severity note;
   end process;
```

WAIT VERSUS TRANSPORT STATEMENTS

One of the most challenging aspects of using HDL simulators is the concept of time. In VHDL, programmers broadly classify code as concurrent or sequential. Multiple sequential blocks of code execute concurrently and use sensitivity lists or wait statements to pace their execution. Within a process, the software schedules events, but simulation goes no faster until the program encounters a wait statement. You can use multiple wait statements to control the sequencing of various signals, but this approach causes the timing of individual signals to interact, complicating their control. Similarly, you can use transport-delay specifications to time signal transitions, but these specifications control the timing of a series of transitions on a single signal, independently of all other signals. You still need a wait statement to speed simulation, but the timing sequence in the transport statement rolls off the simulation-event wheel as time advances due to one or more wait statements. These wait statements may be in any part of the code. It is easier to individually specify

TABLE 2—COMMONLY USED VHDL ATTRIBUTES

Attribute	Description and usage
Signal'event	Returns a true value if the signal had an event (changed value) during the current simulator tick; useful for synchronizing events with a strobe signal. For example, when the WR~ goes high on an SRAM, then a write cycle may have taken place. This time is appropriate for executing the appropriate code in a process that models an SRAM.
Signal'last_value	Returns the value of the signal before the last event.
signal'last_event	Returns the time elapsed since the most recent event on a signal; useful in an SRAM model to see whether the address had been stable throughout the duration of the write cycle.
signal'last_active	Returns the time elapsed since the most recent transaction (scheduled event) on a signal.
signal'active	Returns a true value if any transaction (scheduled event) occurred on a signal during the current simulator tick.
type'image(*expression*)	Creates a text image of the specified expression; useful for text reports; the expression must be of the same type as that to which the attribute was attached; especially useful when used with assert statements.
type'value(*string*)	Returns a value of the attached type.
signal'delayed(*time*)	Creates a signal identical to the attached signal but delayed by time.
signal'stable(*time*)	Creates a Boolean signal that becomes true when the signal is stable for the specified time period; could be used for checking setup-and-hold times.
signal'quite(*time*)	Creates a Boolean signal that becomes true when the signal is quiet (having no events scheduled) for the specified time period.
signal'transaction	Creates a bit signal that toggles when a transaction or event occurs on a signal.
type'pos(value)	Returns the position number of the value for the specified enumerated type; this feature is useful for generating an integer that references the value of an enumerated type.
type'val(integer)	Returns the value of an enumerated type from the specified integer value; the first enumerated value has a position number of 0, and all subsequent values have numbers of 1, 2, 3, and so on.

signal transitions with transport statements, but it is generally more difficult to unravel the timing when stepping through code during debugging.

You can use both methods of controlling signal sequencing and timing in a given testbench. Wait statements are easiest to use when a group of signals repetitively and simultaneously change. This situation is typical of synchronous signals simultaneously changing during every clock cycle. When modeling an asynchronous interface, such as with an Intel-style microprocessor, you may need to individually adjust signals. Using wait statements, which delay the application of all signals downstream, this approach involves a complex problem. Individual transport statements allow you to specify a series of transition delays for each signal, defining a waveform for each signal. Although this technique decouples signal timing, it also makes it more difficult to visualize the sequencing of individual timing. For instance, if a key requirement is to guarantee an address-hold time from the end of a write strobe, it may be easier to use a wait statement to move the simulation-timing wheel forward a fixed time between the unassertions of the write strobe and address bus. You may want to experiment to figure out what fits your preferences and design requirements. Multiple wait statements are easier to step through during debugging than are transport statements, but transport statements are more compact for describing the same behavior. With the modularity of modeling, it is easy to use each mechanism where it best fits.

You can improve this simple model slightly by adding statements that check the responses of the unit under test (**Figure 1b**). Automatically checking data is a mixed blessing in such a simple scheme. It is difficult, with the simple testbench, to have enough control to do all the checking you need to validate the circuit responses. Just sampling the data at the correct time can be difficult. Checking setup-and-hold times and all the other parameters of importance is a daunting task when the program forces all the code for the simulation to reside in one process in a flat model. You can somewhat remedy the situation by setting up multiple processes to check signals, parameters, or interfaces, but again this ad hoc process is difficult to control, manage, and maintain. Because you mix structure and function, it is also difficult to reuse such designs in new applications.

KEY MODELING CHARACTERISTICS

The basic idea behind a testbench based on modeling is to create functional models of all board-level components that surround the unit under test (**Figure 2**). The top-level file usually contains structural VHDL code that ties the various models together as instantiated components. At least one of the components is the unit under test. The other components are functional equivalents of the other devices on the board. In this example, you need to simulate a Lattice CPLD. The design is an interface between a 33-MHz i960RP microprocessor and dual banks of SRAM. It allows the system to access the memory at half the bus speed by alternating access to two banks of SRAM. The CPLD is the unit under test, but you have to model the CPU, clocks, transceivers, and SRAM. The

TABLE 3—USEFUL FILE I/O COMMANDS

File_open*(file,name,mode)*	Opens file object with the specified name for a mode of read, write, or append.
File_open*(status,file,name,mode)*	Opens file object with the specified name for a mode of read, write, or append and assigns value to status of type file_open_status that has a value of OPEN_OK, STATUS_ERROR, MODE_ERROR, or NAME_ERROR.
File_close*(file)*	Closes specified file.
Read*(file,object)*	Reads from specified file into specified object.
Readline*(file,line)*	Read a line from the specified file.
Write*(file,object)*	Writes specified object to specified file.
Writeline*(file,line)*	Writes a line to the specified file.
Endfile*(file)*	Returns Boolean true value if file pointer of specified file is at the end of the file.

TABLE 4—USEFUL TEXT I/O COMMANDS

Additional text-handling commands added by the standard textio package	
Apply_exponent	This procedure reads in numeric characters and the "_" character and uses them as an exponent for the real parameter. It indicates the success of the operation through the ok parameter.
Apply_exponent	This procedure reads in numeric characters and the "_" character and uses them as an exponent for the real parameter. It indicates the success of the operation through the ok parameter.
Apply_fraction	This procedure reads numeric characters and the "_" character from a line variable and converts them into a fractional number. It indicates the status of the conversion throught the ok parameter.
Apply_mantissa	This procedure reads numeric characters and the "_" character until encountering a "." character. It converts these characters into a real number and indicates any problems through the ok parameter.
Extract_integer	Once the optional leading sign is removed, an integer can contain only the digits "0" through "9" and the "_" (underscore) character. VHDL disallows two successive underscores and leading or trailing underscores.
Extract_real	This procedure reads numeric characters and the "_" character until encountering a "." character. It converts these characters into a real number and indicates any problems through the ok parameter.
Grow_line	This procedure increases the length of the specified line by the indicated increment.
Int_to_string	This procedure converts a string to an integer.

SRAM models can check all the signals from the unit under test on every access cycle. You can easily put all the required setup, hold, and pulse-width checks into the SRAM model. It would be difficult to put those checks into a simple structure, especially when you need to include all the other checks into the test, as well. A designer's task is easier with signal checking embedded in the functional models.

One interesting aspect of modeling a functional system involves defining the overall control mechanism. In a simple testbench, a test designer most naturally places the main test process at the top level of the hierarchy, often in a testbench comprising one file. In a testbench based on models, the controller of data flow in a design parallels the controller in the actual system. For intelligent cards, this controller is the CPU or microcontroller. You would probably control slave adapter cards via the system bus. In this example, the i960 processor is the system controller, and the simulation naturally flows through the CPU model. The main control process is not even at the top level of the testbench; it is embedded in the CPU model. This situation may seem strange to those who have used only simple testbenches, but it is a natural consequence of modeling.

In this example, both the memory and the SRAM controller are slave devices. Because the CPU model drives the simulation, a designer needs only to specify the sequence of bus cycles with a method to route data into and out of the design. The CPU model generates all individual bus signals. By handling the simulation at this higher level, a designer can deal with system-level functions and let the models handle the details. Once you generate the models, you can develop a complex functional simulation more quickly than with low-level methods.

Modeling often separates validation of control-path functions from validation of datapath functions. Simple methods merge these functions, again complicating the issues of simulation control. The model generally verifies control-path functions. With an SRAM model, you can easily determine what code needs to go into a model to verify that the control signals are functioning properly. Putting timing checks into an SRAM model allows you to place the checks in one location that is active whenever the model accesses the SRAM. You can implement other control-path-timing and functional verification in behavioral models. You can sometimes internally test datapath functions in the models, as in the case of a CRC generator/checker. Generally, it is more reasonable to check the datapath at a higher level of abstraction. In this case, it is more convenient to have the CPU model perform write- and read-back cycles to verify the operation of the memory subsystem. The CPU model verifies data integrity. You do the verification in much the same manner as writing self-test code for a microprocessor. For a fully functional CPU model, you could use the same code you developed for system self-test with limited modification to run the simulation; the converse also is true.

Fortunately, you need not use arcane methods to start designing modeling testbenches. Unlike object-oriented programming, testbench modeling does not require a wholesale paradigm shift in the way engineers think of system design. Because the structure that the testbench mimics is the hardware design, building a testbench with functional models requires the same analytical methods that engineers have developed through years of working with schematics and lab equipment. You simply replace the components on a board with functional components in a VHDL design and then figure out how to model each component. Planning a simulation is much like planning the early stages of prototype debugging when a hardware engineer typically writes diagnostic code.

Modeling is most advantageous when you adopt it on a companywide basis. You must design, implement, and validate any models that you don't buy. You have to validate models at a low level of abstraction, looking explicitly at detailed function and timing. An engineer using the models to test a design can be the same engineer who validates the models. Because modeling involves the encapsulation of function, this scenario helps prevent self-fulfilling prophecies, in which the testbench designer perpetuates the same faulty assumptions a system's designer makes. You can organize model building, plan it into the design schedule, and divide it among the design-team engineers. You should establish standards to ensure the consistency and quality of the generated models. This approach ensures the existence of a library of prevalidated models that are a valuable resource for enhancing the simulation capabilities and efficiencies of a design team.

If the effort of developing complex models is beyond the capability of an engineering group, it should consider obtaining models from outside sources. Standards compliance often dictates the need for a fully validated and compliant model. All standard-bus models, including PCI, SCSI, VME, and ISA, are available, as are models that intellectual-property (IP) vendors often supply. The Logic Modeling Division of Synopsys offers a variety of bus-interface models that aid in verifying compliance and interop-

TABLE 5—CPU-MODEL-CONTROL INSTRUCTIONS

Address	Specifies address to be used in subsequent read or write operations.
Data	Specifies data to be used in subsequent read or write operations.
Write single	Performs single write cycle.
Enable data checking	Enables checking of data read back during read operations.
Read single	Performs single read cycle.
Number of burst cycles	Specifies the number of burst cycles to run during subsequent burst reads or writes.
Wait cycle	Specifies which cycle to cause a CPU wait cycle during burst reads or writes; specifying "0" disables CPU wait states.
Number of wait cycles	Specifies the number of wait cycles to insert during subsequent burst reads or writes (if wait cycles are enabled).
Write burst	Performs burst-write cycles.
Read burst	Performs burst-read cycles.

erability of standard-bus designs. Many other third-party vendors also supply a variety of cores. You can find an impressive list of noncommercial model sources on the Web (**references 1, 2, and 3**).

MODELING AND ABSTRACTION

VHDL's inventors envisioned the language as a pure modeling language. Early adopters of VHDL for logic synthesis had problems in that many useful modeling constructs were inappropriate for synthesis. Second, VHDL dictates that you implement many of the necessary features that are part of Verilog, such as type conversions, as library functions. In contrast, you can use the full range of VHDL's modeling capability for generating behavioral-simulation models. You can specify complex behavior without adhering to the RTL conventions that synthesis supports. You can even model analog and mechanical systems using VHDL's mathematical functions. (The analog extensions now under consideration for VHDL are primarily extensions of the language's currently available math functions.) Because you need not synthesize the simulation models, you can use many high-level behavioral constructs to simplify functional-model generation. Defining functions according to the components that perform them yields a functional decomposition that encourages a higher level of abstraction. This technique decouples the functional specification from the low-level operational details and simplifies writing the test program. **Tables 1** and **2** summarize many of the features and attributes of VHDL that are useful for generating modeling simulations.

Although not a feature of VHDL per se, most VHDL- and Verilog-simulation packages allow users to interface executable programs to the simulation. These programs, usually written in C or C++, allow users to implement complex behaviors without the speed or space overhead of using VHDL models. For example, modeling a large SRAM model in C allows the operating-system resources to implement data storage. You could model a memory that exceeded the size of available RAM on the PC or workstation if the operating system could cache data and swap it to disk when necessary. Alternatively, you could model the system in C and use the simulator to allow the C program access to the component models in the simulator. This approach could be faster than VHDL for large simulations but would require an environment including both VHDL and C development tools. This approach might be helpful for an engineer working with systems simulated in C before partitioning and handing them off to a circuit designer for implementation.

WHAT IS A CONFIGURATION?

VHDL defines a configuration design unit that is roughly analogous to a part list for a pc-board assembly. You require a configuration because a given entity can involve multiple architectures. A configuration declaration, which the VHDL keyword *configuration* specifies, identifies the pairing or binding of an architecture to each entity. You do not explicitly need configurations unless you specify multiple architectures. Configurations are most useful for simulation; synthesis tools do not generally support them.

USING VHDL, A SYSTEM DESIGNER CAN MODEL A CIRCUIT AT MULTIPLE LEVELS OF ABSTRACTION.

The 1987 VHDL standard's file-I/O functions were somewhat limited. The 1993 version added syntax enhancements and new features to extend the language's file-I/O capabilities. You can use the VHDL file-I/O functions in various ways. Primary uses are retrieving and applying stimulus vectors; storing response vectors and expected responses for comparison; formatting reports; and reading custom test-language files, which the program then parses.

You can use the test-language-file reading with a higher level of abstraction to control the simulation. Because you specify the commands at a functional level, functional models, which respond to each command according to their programming, perform the detailed implementation. This powerful technique drives simulations at a high level. You leverage your system-level knowledge to the greatest degree because you exercise control at a level that is most convenient for checking system functions. When a text file determines control flow, the custom test language becomes specific to a simulation. The VHDL simulation interprets the code. You can quickly generate and check new test cases because you need not compile the new code, because it is not a part of the VHDL testbench. **Table 3** shows the most common basic file-I/O commands. **Table 4** shows the added common commands from the standard text-I/O package.

DESIGNING A TESTBENCH USING MODELS

The IEEE offers resources, including online notes, that are excellent guidelines for writing behavioral models (**references 4, 5, and 6**). The key points are to model a system at multiple levels of abstraction, to hide the system's structure, to focus on behavior and functionality, to ignore timing at the top level of abstraction, to follow standard practices of software engineering, to simplify maintenance and reuse, to structure the design, to define each component to have strong cohesion, to define a loosely coupled set of components, to use top-down iterative refinement, and to use abstract data-typing to hide and encapsulate data.

Using VHDL, a system designer can model a circuit at multiple levels of abstraction. When modeling in VHDL, it is important to follow standard practices of software engineering. Otherwise, the model becomes difficult to maintain, even for the person who wrote it. In addition, to aid the reuse of models, you should carefully and with reuse in mind even create throwaway models. Typical model design and coding practices include structuring the design; iteratively refining a high-level view of the model down to its final form; employing abstract data typing to hide and encapsulate data; and organizing the individual model components so that they are loosely coupled with few interface signals and have strong cohesion, keeping strongly related functions in the same architectural body.

In **Figure 3**, the schematic links the design of a CPLD-based SRAM controller to models for all the other major components in the design. This Ramix Corp CPLD design targets a Lattice M4A5-192/96-7VC part. Ramix chose this part because the speed locking of the internal delays allowed the part to work at an effective internal frequency of 132 MHz.

The design required state changes on both edges of a 66-MHz clock. Using the highest speed would push the usable frequency to around 180 MHz.

The CPU model generates the bus cycles to control the simulation. **Table 5** lists the special instructions that specify the test sequence. The instructions are simple and modeled after the required bus operations. Modeling all the assembly-language operation codes would have been too complex and would have been less useful. The model needed only 10 instructions, and these instructions include setting the address and data for a given operation. With these instructions, you can check all the various modes of reading and writing the SRAM. The testbench uses simple combinations of commands to implement all the required test scenarios. The model has two implementations. The simpler model has the test sequence embedded within the model itself, implemented as VHDL code that you have to compile each time you change the test. This procedure violates your desire to decouple high-level tasks from low-level tasks. In the second procedure, a test language-parsing engine drives the simulation. The text-parsing engine may seem to add an extra level of complexity, but it offers advantages in flexibility and ease of use. Because an external text file includes the test sequence, the program need not compile the sequence before it runs. This feature lets you quickly and easily add test scenarios. In effect, this method offers an application-independent simulation language that you can use in any system that requires this functionality on an i960 CPU.

YOU CAN EASILY CHECK THE LISTING FROM THE BEHAVIORAL-MODEL TESTBENCH. THE LAST LINE TELLS WHETHER ANY ERRORS EXIST.

Several options for implementing the SRAM models exist. You can implement the models as an array, simply storing data in array locations according to the location addresses. Another choice is to build a model that can transfer address and memory data to a disk file for later verification. This example does not require the memory to pass large amounts of data through the design. The engineer designed the memory with two 16-word-deep arrays to store data—one at the lower address space of the model and the other at the top end. The designer implemented the second array to allow RAM access to roll over from one page to another. The design uses the SRAM address to index into the arrays. The model responds to standard write-enable, output-enable, chip-select, address, and data-RAM-control signals. The design stores data in the array upon execution of a legal write cycle and retrieves data after execution of a legal read cycle. In addition to storing and retrieving data, the model can also check SRAM timing requirements. You can select the timing parameters to be none, minimum, typical, or maximum. The minimum access time doesn't matter, so the designer set it to zero delay.

The chip-select decoding model is a simple combinatorial-decoder function with no parameter checking. The model duplicates the function- and propagation-delay timing of the circuit. You can select the delay as none, minimum, typical, or maximum. The latched-transceiver model is a simple sequential model with limited parameter checking for data setup and hold.

The simple function of this circuit requires simple test cases. You accomplish each of the bus-cycle types for the i960 model by writing to the SRAM and reading the data back. The test cases include single write then read, bank 0 or 1 start, burst write then read, bank 0 or 1 start, with or without CPU wait states, with or without SRAM page break, and overlap CPU wait states with page break. These relatively simple cases are sufficient to test the functions of the memory-control CPLDs. The CPU model checks the datapath by verifying that the data written is the same when read back. The SRAM models, which require proper control signals to store and retrieve data, check the control signals. The models have timing checks distributed throughout themselves. The CPU, SRAM, and latched-transceiver models all incorporate code to verify proper signal timing. If a test does not meet required setup, hold, or pulse timings, the model that detected the error reports each error. You can

model.

You can easily check the listing from the behavioral-model testbench. The last line tells whether any errors exist. If there are no errors, your job is done. If a failure occurs, a quick inspection reveals the location, along with a clue about the location and the cause of the failure. The testbench automatically runs all checks each time you run the simulation.□

References

1. Range of VHDL information, www.vhdl.org/comp.lang.vhdl/FAQ1.html.
2. Interactive VHDL tutorial and demo, standards, ieee.org/catalog/press/vhdlmods/tutorial/html/homepage.html.
3. Example code for the models and testbenches in this article, www.vantis.com/literature/literature.html.
4. *1076-1993 IEEE Standard VHDL Language Reference Manual*, IEEE Press, 1993.
5. Pellerin, David, and Douglas Taylor, *VHDL Made Easy!*, Prentice Hall, 1997, ISBN 0-13-650763-8.
6. Quigley, Eamonn, *The VHDL Reference Guide*, Esperan Ltd, 1995.

Acknowledgment

Many thanks go to Saeed Karamooz of Ramix Corp for his support and for allowing me to use Ramix's design as an example for this article.

Author's Biography

Gary Peyrot is a field application engineer with Lattice Semiconductor (www.lattice.semi.com), where he has worked for two years. In his current position, he helps customers design with PLDs and development tools. He received a bachelor's degree in biochemistry and a master's degree in scientific instrumentation from the University of California—Santa Barbara. His hobbies include karate, philosophy, and debating Darwinists.